1995

An approach to implementing multiple inheritance in an object- oriented programming language

Daniel H. Wilson
Lehigh University

Wilson,
Daniel H.

An Approach to Implementing Multiple Inheritance in an Object-Oriented Programming...

October 8, 1995

An Approach to Implementing Multiple Inheritance in an Object-Oriented Programming Language

by

Daniel H. Wilson

A Thesis
Presented to the Graduate and Research Committee
of Lehigh University
in Candidacy for the Degree of
Masters of Science

in
Electrical Engineering and Computer Science

Dr. Edwin Kay (Advisor)
Lehigh University
Bethlehem, PA
May 29, 1995

Certificate of Approval

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

30 May 1995
Date

Thesis Advisor

Chairperson of Department

Acknowledgements

I want to thank Dr. Edwin Kay at Lehigh for turning me on to object-oriented programming by teaching me Smalltalk-80 on an old Tektronix Unix workstation. Its a piece of computer history that I will carry with me for a long time. And also for keeping me honest about what I needed to understand in writing this thesis, and with his teaching in general.

I owe a thank you to Bob Mack at Binney and Smith, Inc. for believing in education enough to support me in pursuing my Masters. It was an unusual thing at Binney and Smith for someone to pursue a very technical Computer Science Master's degree, taking classes during the day. In a very business oriented environment, many times I was asked "don't I feel something lacking in that there is no direct application of what I was learning." But Bob supported me anyway, and for that I am grateful.

Thanks, Mom and Dad, for lending me the cash to cover tuition whenever I was a week or two late for registration.

I owe a thank you to my three boys, Crosby, Elliot, and Noah, with whom I could have spent more time had I not had my nose stuck in a computer monitor, or paper, programming, word processing, reading or writing. I love you guys.

Kathy Magas Wilson has been behind me since I started taking learning seriously in undergraduate school 15 years ago. She brought me tea late at night when I wasn't sure I'd make it through my first Math, Physics and Computer Science classes. She's supported me in the advancement of my education and career more than anyone else. While I work full time and go to school, she's worked and managed the home and family affairs. She's spent many late nights and weekends at home taking responsibility for our children, allowing me the freedom to pursue my Masters. And she's seen the worst side of me, when the stress of trying to do it all has worn on both of us. I dedicate this thesis to my wife Kathy.

Table of Contents

Abstract

It is a given that Multiple Inheritance is an important programming concept, that it models our world more naturally than single inheritance. It has created some controversy over its value as a programming tool, because there doesn't seem to be a clear consensus on how to implement it in a consistent manner independent of the implementation language. Intuitively, we know that we should use multiple inheritance when we program, because it is how we think. However, because there is no one well-understood algorithm for implemention, we tend to shrug it off as too difficult to implement. Alternatively, we may implement it with some over-simplified algorithm that does not truly model our world. This paper will describe a proposed implementation for multiple inheritance. I attempt to do it naturally, without over-simplifying the solution. The end result places a responsibility on the programmer to better understand multiple inheritance so that he will better understand his application. At the same time the language will provide the flexibility to more realistically model our world as we do, in our own thought processes, when we classify the objects within it.

Introduction

I begin by reviewing some basic concepts in object-oriented programming. Some commonly cited references, for those who wish to learn more about object-oriented programing in general, are Goldberg and Robson [9], Cox [6], and Meyer [14].

Then in the first part of this paper, I will ask the question "how can super-classes be combined in a sub-class?" The purpose of this first section is to give a better understanding of how multiple inheritance can be used. Also, I have placed some restrictions on how multiple inheritance should be implemented. It will act as a background definition for the following sections that discuss the issues involved in implementing multiple inheritance.

The two important issues with multiple inheritance are (1) the inheritance search itself, and (2) name collisions between like-named properties (variables and methods) within sub-classes and their inherited super-classes. These two issues will be the subjects of the second and third parts of this paper.

Basic Concepts

James Martin [12] describes the fundamental ideas of object-oriented technology as:

1. Objects and classes (a class being the implementation of an object type)
2. Methods
3. Encapsulation
4. Requests

5. Inheritance

Objects

An object is any thing, real or abstract, within which we store data, along with operations that manipulate the data. In object-oriented analysis and design , we are interested in the behavior of the object. An object type is a category of object. An object is an instance of an object type.

Operations are used to manipulate the data of an object. They can only reference the data structures associated with their own object type. To use the data structure of another object, they must send a message to that object. An object is thus an entity whose properties are represented by data types and whose behavior is represented by operations.

Methods

Methods are operations implemented in software.

Encapsulation

Packaging data and operations together, as within an object, is called encapsulation. The data are hidden from other objects, protecting it from unintended or arbitrary use. By hiding the details of its internal implementation, an object's users understand what operations may be requested of an object, but do not know the details of how the operation is performed. Encapsulation is important because it allows an object's implementation to be changed without requiring the applications which use the object to be modified also.

Requests

A request asks that a specified operation be invoked using one or more objects as

parameters. The operation performs the appropriate method and, optionally, returns a response.

A request is a more general notion than a message, because more than one object can participate in a request. For example a request may ask that a Part be returned to a Bin, leaving the selection of which method within which object (Part or Bin) to the selection mode of the object-oriented implementation. A message is a request to carry out a given operation on a given object. The message that constitutes the request contains the name of the object, the name of the operation, and sometimes one or more object parameters.

Classes

A class is a software implementation of an object type. The term object type is used in object-oriented analysis; the details of classes are determined during object-oriented design. A class declares a data structure and the methods that specify the operations that may be used with that data structure.

Inheritance

A high-level concept can be specialized into lower-level concepts. For example, the object type Person may have subtypes Civilian and Military Person. Military Person may have subtypes Officer and Enlisted Person. There is a hierarchy of object types, subtypes, sub-subtypes, and so on.

More Background on Inheritance

As I have said, a class implements the object type. A subclass inherits properties and

behaviors of its parent class; a sub-subclass inherits properties and behaviors of the subclass, and so on. It inherits data types and methods. It also has methods and sometimes data types of its own. Sometimes a class directly inherits properties and behaviors of more than one immediate parent superclass (as distinct from a superclass of a superclass). This is called multiple inheritance.

Categorizing objects

James Martin [12] points out that an object type is a category of object. Categorizing objects into object types creates the conceptual building blocks from which a system can be designed. In fact an object can be categorizied in more than on way. In figure 1 one person may regard the object, named Kathy, as a Woman. Her boss regards her as a Program Director. The social workers whom report to her consider her a Manager. The local PTO recruits her as a Volunteer. Her children know her as Mom; her husband counts on her in more categories than can be enumerated, and so on.

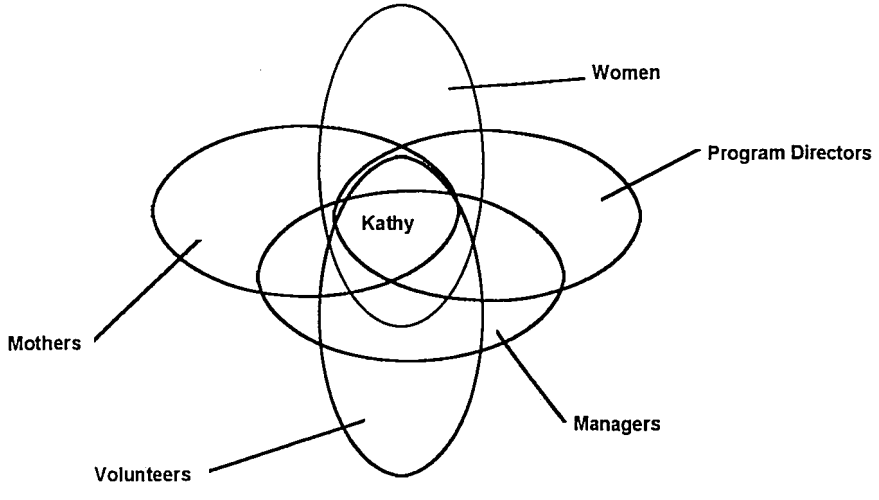


Figure 1: The same object can be categorized many ways.

Generalization Hierarchies

Humans organize knowledge by arranging categories into hierarchies. The categories at the top of the hierarchy are more general types of objects than those further down in the hierarchy. This concept is called a generalization hierarchy. Generalization is the result (or act) of distinguishing an object type as being more general, or inclusive, than another. Everything that applies to an object type also applies to its subtypes. Every instance of an object type is also an instance of its supertypes.

Fern Diagrams

A generalization hierarchy can be represented by a Fern diagram. Fern diagrams are sometimes networks rather than tree structures, because a subtype can inherit properties from more than one supertype. A fern diagram usually progresses from left to right, with no arrows. Object types inherit properties of the supertypes on their left. The fern diagram does not depict what is inherited. Fern diagrams are useful to help us think more clearly about good categorization. They also show the inheritance paths that will be implemented in class hierarchies. Figure 2 shows a fern diagram of creature categories. Some creatures have multiple supertypes and inherit properties from these supertypes. For example, a Whale has properties of an Aquatic Creature and a Mammal.

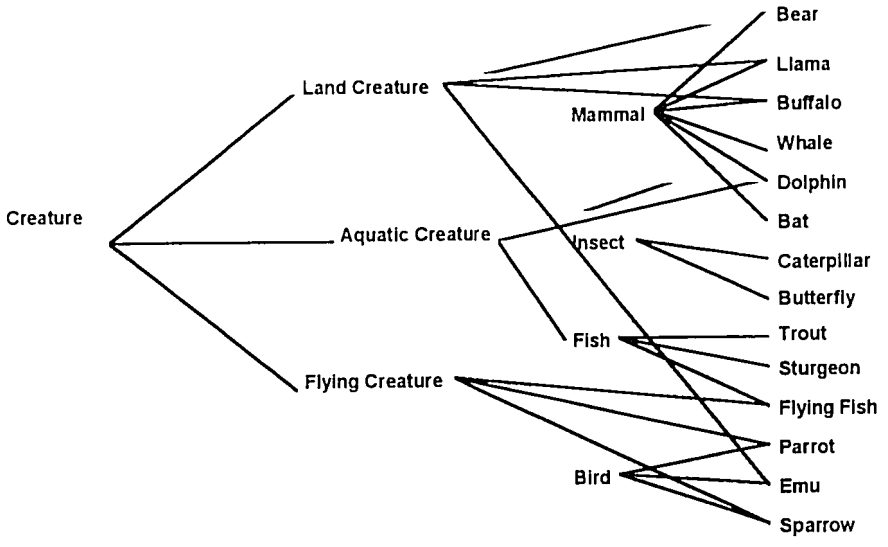


Figure 2: A Fern diagram showing a categorization of creatures.

Class Inheritance

Generalization, as we have said, is a conceptual notion. Class inheritance (usually referred to simply as inheritance) is an implementation of generalization. Generalization states that the properties of an object type apply to its subtypes. Class inheritance makes the data structure and operations of a class physically available for reuse by its subclasses. Inheriting the operations from a superclass enables code sharing - rather than code redefinition - among classes. Inheriting the data structure enables structure reuse.

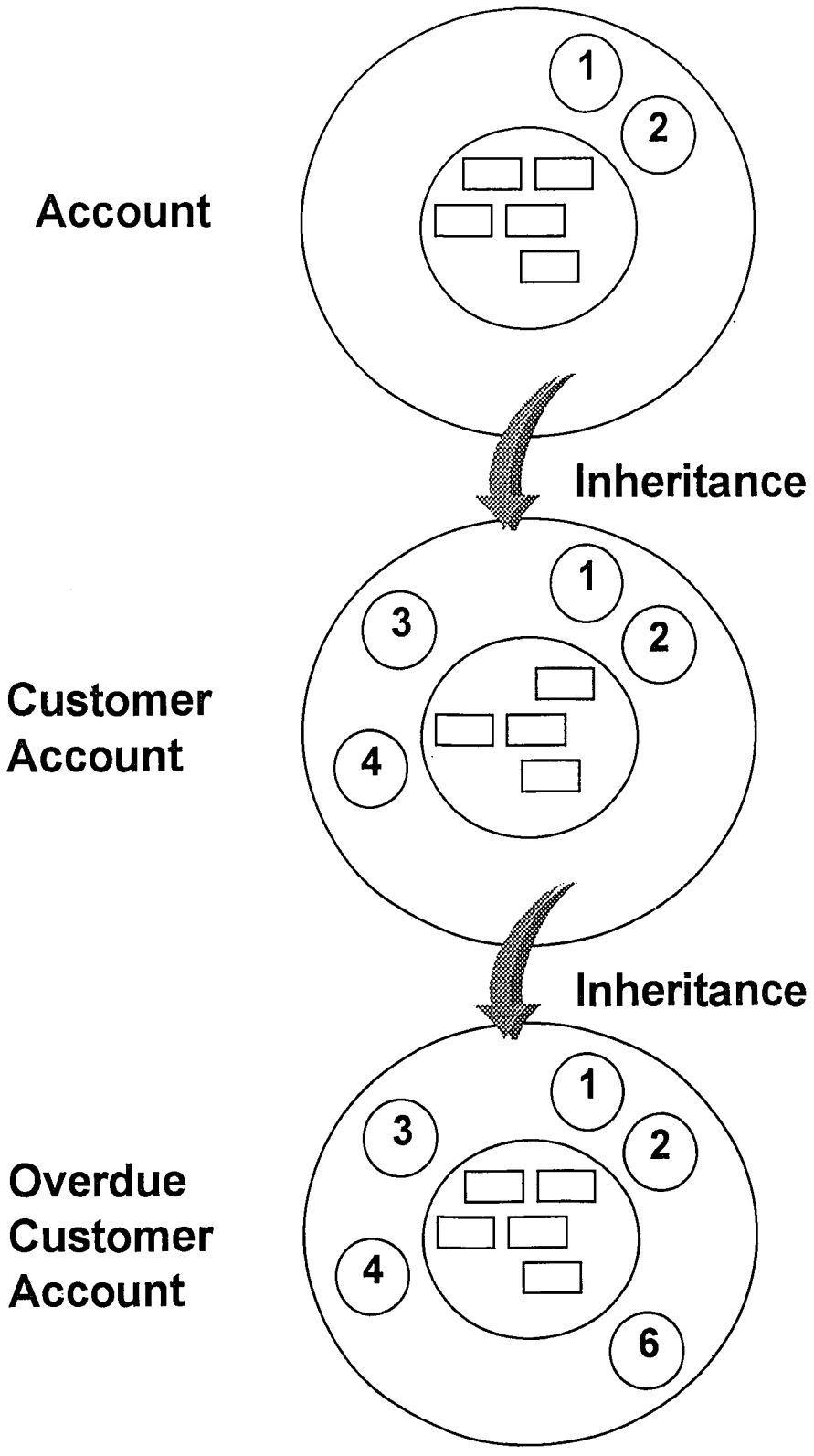


Figure 3: Inheritance.

In figure 3 the class Customer Account inherits methods 1 and 2 from the class Account. Customer Account has two methods of its own: 3 and 4. The class Overdue Customer Account inherits methods 1, 2, 3, and 4, from the class Customer Account - while having one method of its own - 6.

In multiple inheritance, a class can inherit data sructures and operations from more than one superclass. Single inheritance is shown in figure 3, while multiple inheritance is shown in figure 4. In object-oriented analysis, the analyst indicates that Overdue Customer Account has two supertypes - sharing the common supertype Account. In object-oriented design, the generalization hierarchy is implemented using inheritance. Overdue Customer Account inherits the features from classes Customer Account and Overdue Account. Therefore, Overdue Customer Account has the following operations physically available for reuse - 1, 2, 3, 4, 5, and 6.

The classes high in the hierarchy are most important, because they are employed multiple times in multiple subtypes. If they are poorly designed, their lower level subtypes will be poor. The design of these high level classes should be reviewed carefully.

Inheritance is involved in selecting a method when a request is sent to an object. If when a request is sent to an object, the list of permissible operations for that object is checked, and the requested operation is not in it, the object-oriented implementation then automatically checks the superclass of the object. Should the operation not be found in that superclass, the inheritance selection mechanism would continue its search through all the object's superclasses, level by level. If found, the operation would be selected. If not found, the source of the request would be regarded as invalid. Object-oriented

programming languages, such as Smalltalk, detect these invalid requests at runtime. Object-oriented programming languages, such as C++, resolve these requests at compile-time so that no invalid requests can occur.

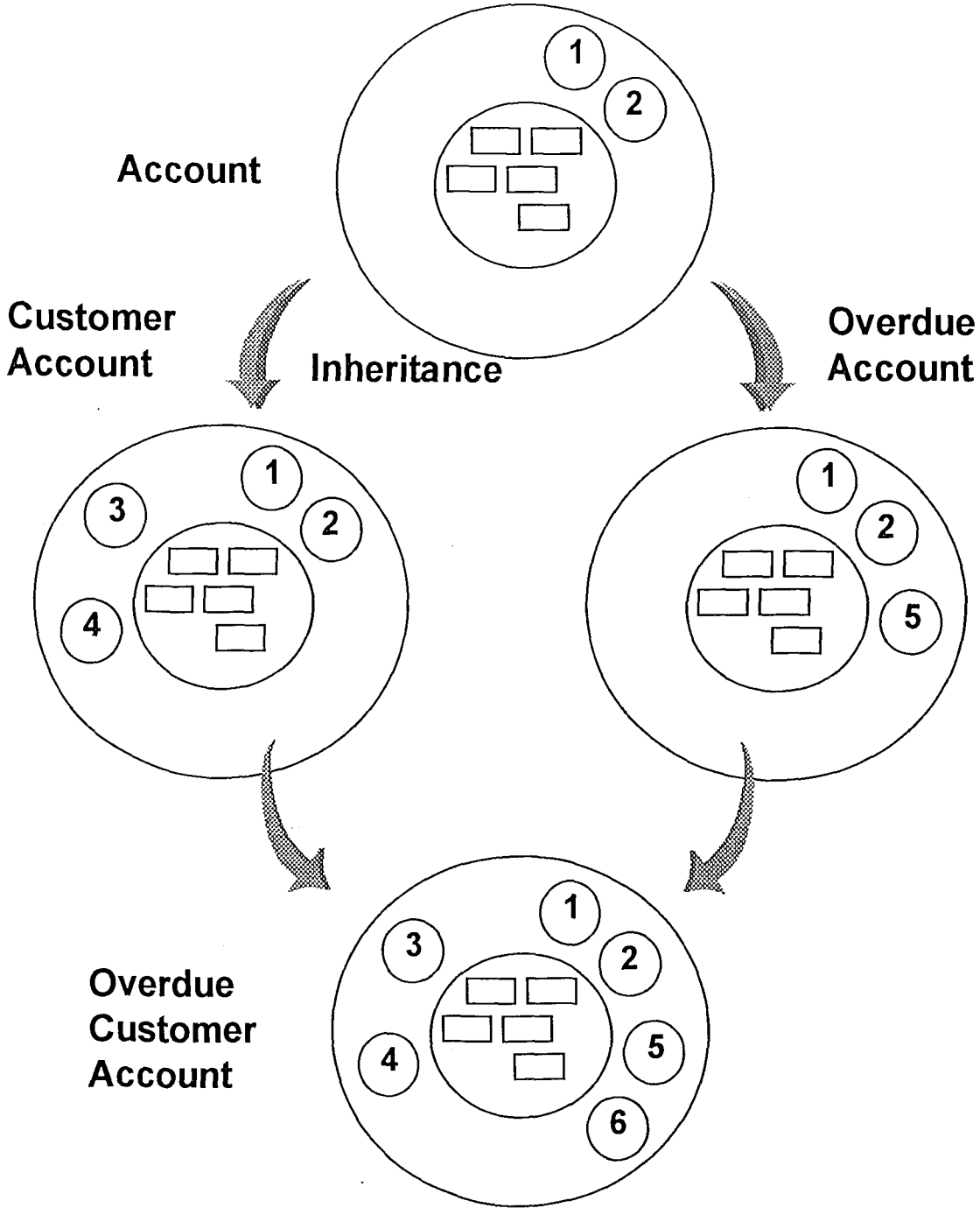


Figure 4: Multiple Inheritance.

Inheritance, then, allows a class to reuse the features of its superclasses. In this way, users need only specify what should be done - leaving it to the selection mechanism to determine how the operation is located and executed. The selection mechanism shifts the burden of locating the correct operation from the source of the request to the object-oriented application.

How can super-classes be combined in a sub-class?

I want to be clear here about what it is I'm proposing to implement. Snyder [18] raises some issues about encapsulation and inheritance. In doing so, he makes clear some important points about how inheritance should work with instance variables. I'll review those points in an effort to clarify how super-classes can be combined in a sub-class.

Perhaps the most basic point is that object-oriented programming requires a client to use operations to access an object's instance variables. Clients do not have direct access to an object's instance variables.

Two categories of clients

In an object-oriented language, a class (implementation of an object) has two types of clients. One category consists of clients that instantiate objects of the class and perform operations on them. The other category consists of clients (other class definitions) that inherit from the class.

Instantiating

System designers often use inheritance to construct families of software components. These software components are implementations of stand-alone objects, or "base classes", where a "base class" is one that defines a complete set of operations and is generally designed to be instantiated.

Inheriting

Designers also create classes to provide features to be inherited by other classes. Inheritable components are commonly provided by application interface tool kits; a class Window, for instance. Such a class is called an "abstract class". An "abstract class is not meant to be instantiated, but rather its sole purpose is only to be inherited. An example of an abstract class would be Land Creature from figure 2.

In multiple inheritance sub-classes can inherit from multiple "base classes". This is the situation that typically leads to shared ancestors. Later in the paper we will provide an example of a sub-class inheriting from two "base classes" -- University Secretary will inherit from University Employee and Secretary. An alternative approach encouraged by Snyder is to inherit from one single "base class" and one or more "mixin classes". A "mixin class" defines a set of operations related to one particular feature and, like an "abstract class", is designed only to be inherited from ("mixed into") a class. An example of combining a "base class" with an abstract class can be taken from figure 2. A Land Creature was combined with a Bird to give te sub-class Emu. Overdue Account in figure 4 can be thought of as a "mixin class". Combining it with the "base class", Customer Account, gives us Overdue Customer Account.

Encapsulation

Clients of an object that is instantiating a class do not have direct access to that object's instance variables (they are only accessible through operations of the class). Inheriting clients, on the other hand, expose instance variables of the parent to violations of

encapsulation in some languages (i.e., Smalltalk). The designer of a class often wants to give greater access to inheriting clients than to instantiating clients. Smalltalk, for example, responds to this need by granting the inheriting client full access to the instance variables defined by the class. What happens when a parent changes (i.e., deletes, adds or renames) an instance variable that is inherited by its descendants? The descendants must be recompiled, and, we hope, are not adversely affected. The designer can no longer rename, remove, or reinterpret an instance variable without the risk of adversely affecting descendant classes that access the instance variable. Snyder [18] proposes that, to preserve the full benefits of encapsulation, the external interfaces of a class should not include instance variables. Instance variables would be protected from direct access by users of an object by requiring the use of operations defined by the class to access the state of the object. Similarly, descendants would be required to use operations provided by the class to access inherited state. In essence, instance variables would not be directly inherited, only operations that give the effect of direct access to instance variables would be inherited. This has the advantage that instance variables in the super-class can be changed without requiring changes in its descendants that have been designed based on the super-class. This approach has The disadvantage that it requires the programmer to write code to access the instance variables in a class. The advantage outweighs the disadvantage in the interest of long term reliablity and maintainability.

Using Parent Operations

Object-oriented languages must provide language support to permit operations to be used effectively by descendant classes. Smalltalk uses "self" to designate the object itself as the receiver of an operation. Sometimes it is necessary to directly invoke an operation in a

parent class that is redefined by the invoking class. Smalltalk provides a mechanism in the context of single inheritance: the pseudo variable "super". Performing an operation on "super" is like performing an operation on self, except that the search for the operation to invoke starts with the parent of the class in which the invocation appears instead of with the class of self. Equivalent features using compound names (ancestor and operation) to specify the desired operations are provided by several object-oriented languages that support multiple inheritance (including Extended Smalltalk).

Snyder's proposal to not grant direct access to ancestors' instance variables restricts the ability of an instantiated class to access an inherited state. An additional responsibility falls on the designer's shoulders to provide the appropriate operations to access the inherited state of a class when it is created. As said above, this is a good thing. No access to an inherited state is possible unless the appropriate operations have been provided by the defining ancestor class (and all intervening classes). This also means that compound selectors (ancestor and operation) can not be used to short circuit the inheritance search. The compound selector can only be used to distinguish like named attributes between immediate parents.

One problem with this scenario is that the operations defined on a class for the benefit of its descendants are not necessarily appropriate for users of instances of the class, yet they are publicly available. A convenient solution, and one that I will adopt from Snyder for our proposed design, is to declare that some operations are available only for direct invocation (on "self") by instantiated descendant classes. They are not part of the external interface to users that would instantiate the object. Such operations are sometimes referred to as "private operations". This is how our proposed design would pass on

inherited state. An operation will be "private" by default. The motivation for this is to enhance maintainability of the programs written in the language. A maintenance programmer who modifies a class will be ensured that unless an operation was defined as "public", he will be free to change its implementation to whatever he feels is appropriate - without repercussions.

Purpose of Inheritance

Snyder [18] puts forth the fundamental question of the purpose of inheritance in two views. One can view inheritance as a private decision of the designer to "reuse" code because it is useful to do so; it should be possible to change such a decision easily. Alternatively, one can view inheritance as making a public declaration that objects of the child class obey the semantics of the parent class, so that the child is merely "specializing" or "refining" the parent class. In the first view one is inheriting "implementation"; in the second view, one is inheriting "external specification" as well.

The first view, that being able to use inheritance without making a public commitment to it in the external interface of a class, is valuable. This view requires a closer look at issues involved with using object-oriented features such as "mixins" and "abstract classes". However, it is beyond the scope of this paper to discuss the issues involved with using mixin and abstract classes. I'll leave it, acknowledging that a language would be too weak without supporting these features, but not committing to explain them in detail. I believe that the following sections dealing with inheritance searches and name collision will provide solutions that will work equally well with both views of the purpose of inheritance.

Most object-oriented languages promote inheritance as a technique for specialization, and do not permit a class to "exclude" an inherited operation from its own external interface. However, excluding classes is something that becomes necessary when viewing inheritance as an implementation technique. Excluding operations is both reasonable and useful. (See Snyder [18] for a discussion about why.) Our discussion on inheritance search and name collision will not attempt to explain how to incorporate this feature. However, I acknowledge this to be an important feature for reusablity that should be part of a multiple inheritance implementation.

Attribute Visibility

To avoid exposing the use of inheritance by a class, clients of the class must not refer directly to ancestors of the class. Specifically, a class may refer to non-immediate ancestors only if they are exposed via the intervening classes.

As mentioned above, it is useful for a class to be able to invoke an operation defined by a parent. We will see later that it is necessary to do so with multiple inheritance to resolve name conflicts. In most languages that support this feature, the desired operation is specified by a compound name consisting of the name of the parent class and the name of the operation. This solution is sufficient: to access an operation of a more distant ancestor without violating encapsulation, that operation must be passed down via all intervening ancestors including at least one parent. Trellis/Owl and Extended Smalltalk allow a class to directly name an operation in a non-immediate ancestor. As a result, the names of ancestor classes are exposed to descendants in these languages. C++ allows a

class to directly name an operation of a non-immediate ancestor only if permitted by the intervening ancestors; thus, the exposure of inheritance can be controlled.

This section has outlined some ground rules for what kind of multiple inheritance I will design an implementation for in the remainder of this paper. I approached it by trying to answer the question "how can super-classes be combined in a sub-class?" I used Snyder's discussion about encapsulation and inheritance to highlight important approaches to using multiple inheritance. I chose a model (functional requirements) for the multiple inheritance for which I intend to propose a design.

The Inheritance Search

Snyder [18] summarizes three common strategies for implementing a multiple inheritance search mechanism. They are: linear implementation, tree implementation, and graph implementation. Snyder recommends the tree implementation. Graph implementation, as proposed by Ducournau and Habib [7], is the one that will be proposed here as the best solution. Please refer to [18] for a discussion of the other two implementations. I'll elaborate on Ducournau and Habib's research [7] for doing an inheritance search. Singh [17] talks about "repeated inheritance" as being a problem that an implementation of multiple inheritance must deal with. Ducournau and Habib [7] handle this problem by ensuring that each node is only visited once, with the unique root of the inheritance being visited last. Also I want to talk about how a programmer will decide what super-class a sub-class will inherit from first. This seems to be important with regard to polymorphism. Ducournau and Habib [7] introduce an idea called "multiplicity" that provides a solution for this problem.

Ducournau and Habib [7] study multiple inheritance using partial ordered sets and graph theory. They distinguish between two main aspects in an inheritance mechanism: an operational semantic (how it works), and an inheritance algorithm (how the inheritance is computed).

An Operational Semantic (How it works)

They first define some useful graph and ordered set notation. We need this because our next step will be to use it to define multiple inheritance.

Graphs and Ordered Sets Notation

Let X be a finite set and R a binary relation on X. G = (X,R) denotes the graph of this relation. Elements of X are called *vertices* and those of R, denoted by (x,y) are called *arcs*. Furthermore x and y are respectively the origin and extremity of this arc, y is also called a *successor* of x and R(x) denotes the set of all successors of x in G.

A *path* $[x_1,\ldots,x_k]$, from x_1 to x_k, of length k>1, is an ordered sequence of vertices such that For all i an element in [1,k-1], (x_i,x_{i+1}) is an element of R. This path is a *cycle* if $x_k = x_1$.

An arc, (x,y), an element of R, is a *transitivity* arc if there exists at least one path of length strictly greater than one going from x to y in G. Such an arc is also called redundant. An arc of the form, (x,x), an element of R, is a *reflexive* arc.

When G is *acyclic* (without cycle) then its transitive and reflexive closure yields a partial order (*poset* for short) denoted $\leq_R$.

For Y a subset of X, R/Y denotes the restriction of R to Y, and G/Y = (Y,R/Y) the induced subgraph. By restriction I mean R_y = {(x,y), an element of R : x,y are elements of Y}. Finally R^d the *dual* of R, satisfies (x,y) an element of R iff (y,x) is an element of R^d.

Multiple Inheritance and Partially Ordered Sets

We now describe important definitions, principles, and properties of multiple inheritance using this framework of partially ordered set theory. Assume that an inheritance system yields a partially ordered relation on a set of objects. Definition 1 gives a formal definition of an inheritance graph. It is followed by a less formal explanation of the terms used within the definition.

Definition 1: An inheritance graph is a *directed acyclic graph* $G = (X,H,\omega)$ where X is the universe of objects, H the inheritance relation with which transitive closure is a partially ordered set, and ω is the unique anti-root (or root in the inheritance terminology) of H.

What is meant by the inheritance relation, H, imposing a binary relation on the vertices of X, is that the vertices are sub-class, super-class object pairs. In object-oriented systems terminology, ω is called the universe root, and if (x,y) is an element of H (resp. $x \leq_H y$), y is called the *father* (resp. an *ancestor*), and x, in both cases, is called a *descendant* of y. Then ω is the ancestor of all the vertices in X. It has no father. In Smalltalk-80 ω would be the class Object.

Definition 1 describes the inheritance search as a graph, made up of vertices that represent the classes in the hierarchy. The classes are related to one another (represented as arcs). The inheritance relation is one between descendants and ancestors.

For a given object, determine its inheritance.

Definition 2: For an object a an element of X, its inheritance graph (also called its hierarchy) is $G(a) = (X_a, H/X_a)$, where $X_a = \{x : x \text{ an element of } X, a \leq_H x\}$, the set of its ancestors (including a itself). In partial order terminology X_a is the upper ideal of a.

Now that I have defined the inheritance graph, and how to determine the inheritance of a given object in the hierarchy, I can state Ducournau and Habib's first inheritance principle:

Principle 1: An inheritance mechanism must follow the inheritance partial order. In particular, the root ω is always the last vertex to be considered.

The following definition describes how properties are inherited(or not) following the partial order that makes up the search on the inheritance graph.

Definition 3: An inheritance search is a total order $\mathcal{P}_a$ on X_a.
An object inherits a property P, with respect to an inheritance search $\mathcal{P}a$ in the following way:

1. P(a) is the first value encountered following $\mathcal{P}_a$, in a vertex of X_a that admits this property, if there is such a value for P.

2. If there is no such vertex, then a doesn't have the property P.

If x and y both have the property P, and $x \leq_H y$ (x comes before y in the inheritance partial

order), then P(y) is hidden by P(x). This is a formal definition of polymorphism. Another way of loking at this is that property P of an object is a default value for all its descendants in the inheritance graph, which admit (don't exclude) this property.

Definition 4: The inheritance mechanism is a mapping : $\wp$: a is an element of X -> $\wp_a$.

If a is an object in the hierarchy, then it is included in the total order of the inhertance search. In order to build this mapping I have to be precise about how it works.

Property 1: The inheritance searches must be stable under redundancy.

Property 1 tells us that every every search must perform consistently, producing the same partial order every time it is executed. If there are redundancy arcs (transitivity arcs), a partial order will include them the same way every time it is generated.

Principle 2: (uniformity)
The inheritance is a uniform mechanism and its searches apply identically for all object properties.

Principle 2 says that the search will be the same for different properties inherited by a descendant from its ancestor. The search will work the same way for one object when it inherits from its ancestors as for another object when it inherits from its ancestors.

Multiplicity

Multiplicity is a way for a programmer to specify in multiple inheritance that one will inherit from one father before another father. It is proposed here to acknowledge that when one is defining father classes, the language may want to provide a facility for the programmer to specify that the sub-class is more like one father than another. If an object a, which is an element of X, has two immediate super-classes (fathers), b and then c, in the inheritance graph, it is natural to interpret this order as a priority: *a inherits more from b than from c.* Object a is more like b than it is like c. In other words, when two fathers exist, the programmer who designed the system more naturally associates one father (b) with a than the other (c). Therefore, the implementation of multiple inheritance may provide a facility to enable this relationship using graph theory. Figure 5 shows an example of an inheritance graph with objects a, b, and c. The left hand side of the figure describes the inheritance relation between the classes. The right hand side shows the multiplicity relation between fathers b and c, that is, inherit from b before c. It also shows the multiplicity between d and e, which I will discuss in a moment. Our algorithm, that I will establish at the end of this section, will handle this facility.

Definition 5: $\mu(a)$, the *multiplicity relative to a*, is the total order relation in H(a), the set of fathers (immediate successors -- see first paragraph of Graphs and Ordered Sets Notation section) of a. The multiplicity is the mapping, μ: $a \in X \rightarrow \mu(a)$. If a has more than one father, say b and c, and there is a binary relation between a and b, and a and c expressed as H(a), where a is an element in X. Then there is a mapping between the vertex a and the relation between b and c called the multiplicity relative to a -- $\mu(a)$.

Besides our graph (and the mapping of objects inheriting properties from it) representing just the inheritance hierarchy, it can also represent the relationship between multiple super-classes within the hierarchy. This multiplicity mapping provides the graph that needs to be traversed when we want to show that object a will inherit a property from father b before father c (again, refer to figure 5).

This notion is implicit in every graph or partial order set representation. Moreover it is used in every graph traversing algorithm, so Ducournau and Habib propose to use it as another programming tool in their definition of an inheritance algorithm. Thus an inheritance graph is $G=(X,H,\omega,\mu)$. M_a is the union of all transitive closures of $\mu(x)$ for x in X_a and $M=(X_a,M_a)$ is the resulting graph. There can be transitivity arcs within this graph. Therefore, M_a is the graph that contains all the arcs that can make up the total order of the inheritance search.

Principle 3: (inheritance versus multiplicity)
In any case the inheritance relation excels the multiplicity..

Excels means that the inheritance relation is more important than the multiplicity. A programmer, when creating a sub-class of multiple parent classes, may decide to define a multiplicity relation between two parents. This would be acceptable as long as the multiplicity does not introduce a cycle into the graph M. It really wouldn't make any sense to have a cycle, anyway. This would undoubtedly be something that was done in human error. If multiplicity introduces a cycle into the graph M, then we can not allow the multiplicity to occur. The object-oriented programming language would flag this as an error.

For an object a, we say that the multiplicity contradicts the inheritance when the graph $G(a) \cup M=(X_a,H/X_a \cup M_a)$ has a cycle. This may happen in many cases as can be seen in figure 5. The inhertance relation is the left side of the figure, including vertices a through e. The right side of the figure shows the multiplicity beween fathers of a -- b and c, and the fathers of b and c -- d and e. The contradiction lies in trying to define a multiplicity between d and e where d should be inherited before e, as well as e should be inherited before d.

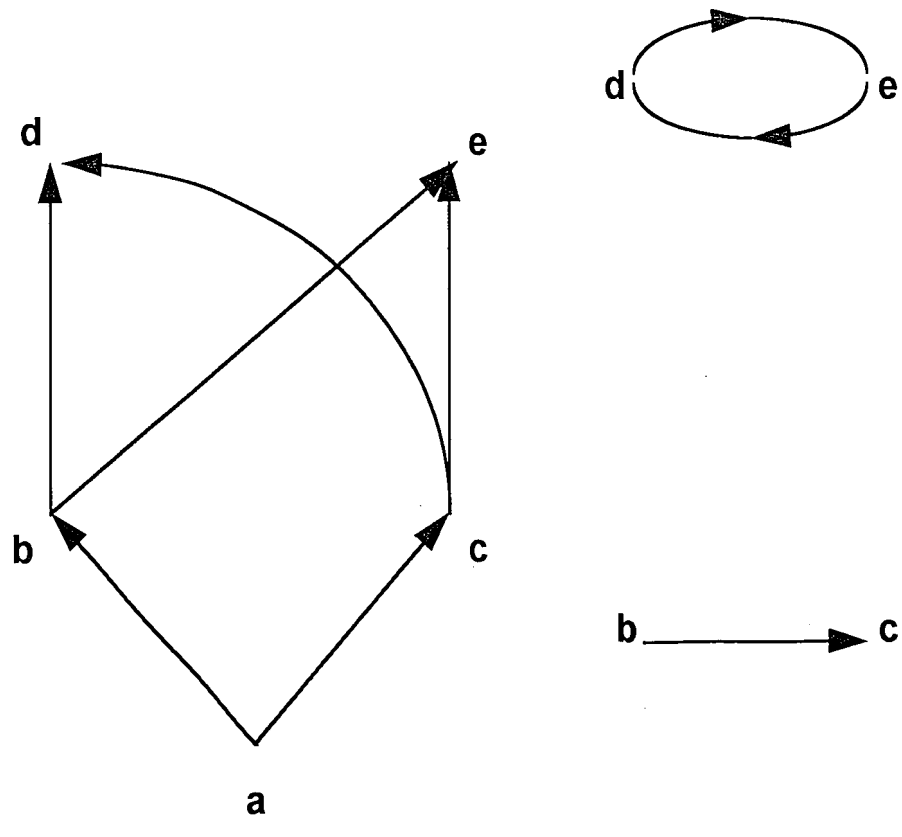


Figure 5: Contradictions between multiplicity and inheritance.

Principle 4: When there is no contradiction between multiplicity and inheritance, the

inheritance search must follow the partial order H U M_a yielded by the graph G(a) U $M=(X_a,H/X_a \text{ U } M_a)$.

The above principle generalizes the first principle (an inheritance mechanism must follow the inheritance partial order) to multiplicity. This is just saying that, if we provide the facility to inherit from one father before another, there can be no cycles introduced into the resulting graph, M.

An Inheritance Algorithm (How inheritance is computed)

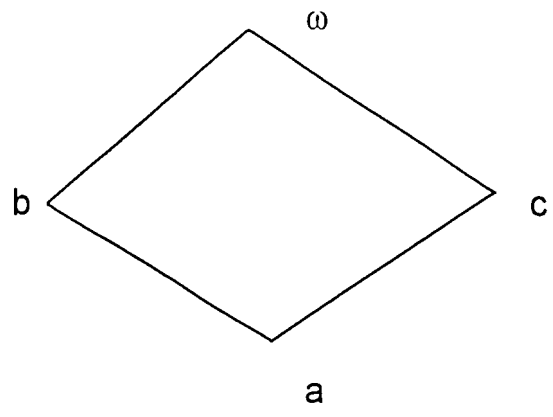
Linear Algorithms

Some Known Inheritance Searches

Most object oriented languages that allow multiple inheritance use a depth-first search of the inheritance graph for an inheritance search. Others use a breadth-first search technique. Yet others use some composition of these two searches. Unfortunately, using well known depth first or breadth first traversing techniques for inheritance searches violates Principle 1, since the universal root ω is not considered last.

Figure 6 illustrates a a depth-first search strategy according to multiplicity. It always violates principle 1. A depth-first search will first search a, then either b or c, then then ω, because by definition it will choose a child of a, and then a child of a's child, and so on until it reaches a node that has no children before it returns back up the tree towards a to search other children. Therefore, it will always reach ω before it has reached all the other

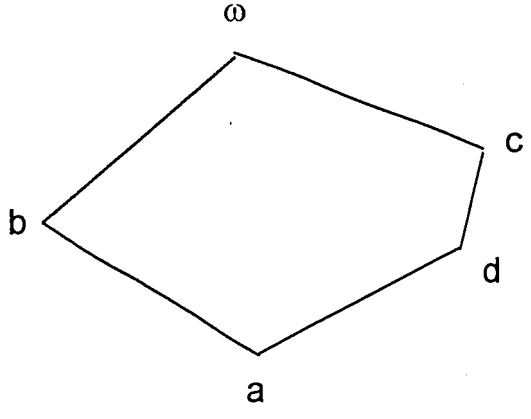
nodes in the graph.



Depth First Search: $\mathcal{P}_a$ = [a,b,ω,c]

Figure 6: Depth-First search contradiction of Principle 1.

Figure 7 illustrates a breadth-first search according to multiplicity. Similarly, by adding a new vertex d, a graph is provided as a counter-example to the operational semantic. This search violates principle 1.



Breadth First Search: $\mathcal{P}_a$ = [a,b,d,ω,c]

Figure 7: Breadth-First search contradiction of Principle 1.

A breadth-first search will first visit a then all the immediate children of a, and then all the immediate children of one of the children of a, and so on. If vertex d had not been inserted into the graph (see figure 7), the partial order would have been [a,b,c,ω]. And that would not have violated Principle 1. Therefore, breadth-first may work with some inheritance graphs, but it cannot be guaranteed to work with all. (Note: A breadth-first search might of figure 7 might also produce [a,b,c,d,ω].)

Linear Extensions

Since, as Ducournau and Habib have pointed out, the known search strategies fall short with multiple inheritance, they suggest using partially ordered set theory to analyze inheritance problems. Towards this aim some algorithmic concepts in partial ordered sets are introduced.

Definition 7: Let $P=(\leq_p,X)$ be a poset:

1. A total order τ on X, is a *linear extension* of P, if $x \leq_p y \rightarrow x \leq_\tau y$.

2. A linear extension τ of P is *greedy* (resp. *depth-first greedy* (*dfgreedy* for short)) if it can be obtained by application of the following rules:

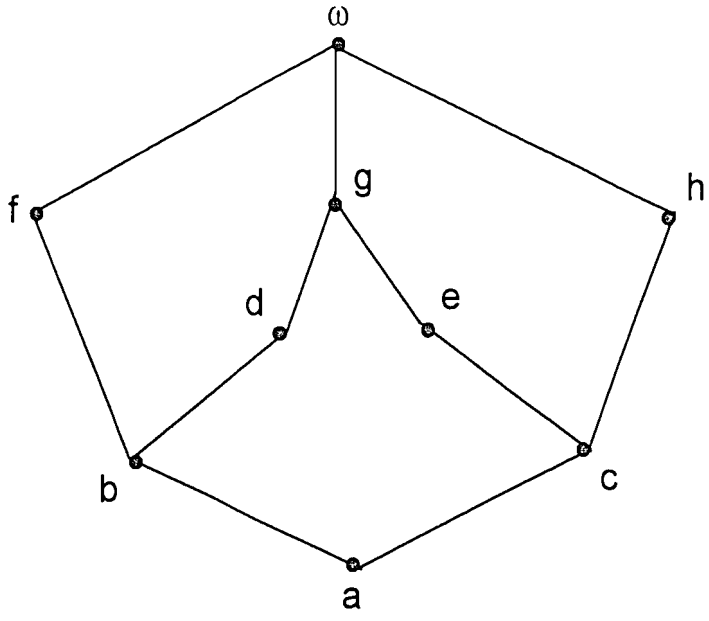
a) Choose for x_1 any minimal element of P.

b) If $x_1,...,x_i$ is greedy (resp. dfgreedy), then choose for x_{i+1} any minimal element of $P_i = P - \{x_1,...,x_i\}$ covering x_i, (resp. covering x_k where $k >= i$ is the greatest subscript possible), if there exists one, otherwise choose any minimal element of P_i.

A minimal element in Pi is some vertex x_{i+1}, such that there exists an arc (x_1,x_{i+1}), or (x_2,x_{i+1}), ..., or (x_i,x_{i+1}). Covering means that the partial order, P_i , that remains to be traversed from x_i, is one that is greater than the partial order that includes $\{x_1,...,x_i\}$. Also, for the greedy part of the definition, there must be an arc (x_1,x_{i+1}), or (x_2,x_{i+1}), ...,

or (x_i,x_{i+1}). There must be an arc (x_1,x_k), or (x_2,x_k), ..., or (x_i,x_k) for the dfgreedy definition.

The first part of the definition of the algorithm says that there is a total order, τ, on the set of all nodes (vertices or classes). It is a linear extension of poset P, if when there is a partial order between two nodes x and y, this implies that the arc (x,y) is also part of the total order τ. The second part of the definition describes when the linear extension is either greedy, or depth-first greedy. Picking the vertex that is a minimal element (an arc exists to that vertex from a vertex that we have already traversed) will enable the greedy portion of the algorithm to work. The depth-first greedy definition relies on the subscripts assigned to each vertex. Choosing the greatest subscript possible for all the vertices that have not yet been searched will result in the best choice according to the dfgreedy definition. In other words the rule "take a minimal element and climb as high as you can" yields the greedy linear extensions. Figure 8 gives some examples of such linear extensions.



[a,b,c,h,e,g,ω,f,d] is not a linear extension

[a,b,c,d,e,f,g,h,ω] is a linear extension

[a,b,f,c,h,e,d,g,ω] is a greedy linear extension but not dfgreedy

[a,b,f,d,c,h,e,g,ω] is a dfgreedy linear extension

Figure 8: Greedy and dfgreedy linear extensions.

[a,b,c,h,e,g,ω,f,d] is not a linear extension because it doesn't conform to the first part of Definition 7. $a \leq_p b$, $b \leq_p c$, $c \leq_p d$, $c \leq_p h$, $c \leq_p e$, $e \leq_p g$, $g \leq_p \omega$, but it is <u>not</u> the case that $\omega \leq_p f$. Therefore the path is not a linear extension. ω must be the last vertex in the partial order.

[a,b,c,d,e,f,g,h,ω] is a linear extension because it adheres to the first part of the definition. $a \leq_p b$, $b \leq_p c$, $c \leq_p d$, and so on.

I will provide algorithms for both the greedy and dfgreedy examples.

A greedy linear extension must first attempt a greedy algorithm, but not by abandoning the fact that it must also be a linear extension. [a,b,f,c,h,e,d,g,ω] is a greedy linear extension, but it is not dfgreedy. It adheres to the first part of the definition, making it a linear extension. Starting with a, the minimal elements from which to choose were b and c. We picked b for no good reason. c could have been picked just as well because is also a minimal element of P_i. Once b was picked, now there were three minimal elements - f, d, and c. It chose f. Now there were still three minimal elements - ω, d, and c. It couldn't choose ω next because then it would no longer be a linear extension. This is where the path distinguishes itself from a dfgreedy algorithm. It doesn't traverse the graph in depth-first order. It chose vertex c, rather than b's right child, d.

A dfgreedy linear extension must first attempt a depth-first search, but must not abandon the greedy algorithm or the fact that it must also be a linear extension. [a,b,f,d,c,h,e,g,ω] is dfgreedy, because it adheres to the entire definition. It is a linear extension, from part 1. It is depth-first, and it is greedy. Its depth-first choices from a are b or c, as are its greedy choices. From b, it must choose f or d to remain depth-first. It chose f. It did not choose ω, because that would have violated Principle 1, which would no longer allow it to be a linear extension. Then it had to choose d to remain depth-first. At that point its minimal elements were ω, g, and c. It could not choose ω, or g, because it would no longer be a linear extension if it did. It chose c. Now the minimal elements were ω, g, e and h. Again, it could not choose ω, or g, because it would no longer be a linear extension if it did. It could choose e or h to remain depth-first. It chose h. Now the minimal elements were ω, g, and e. It had to choose them in the order e, g, and ω to remain a linear extension.

Inheritance Algorithm

I'll now present an algorithm from Ducournau and Habib [7] that uses the framework presented in previous sections on an operational semantic of the inheritance search and our discussion of linear extensions. It is shown [7] to be of good algorithmic complexity (linear-time).

A depth-first search of a graph can be represented as a stack of the vertices as they are traversed within the graph. The terms dfi, and dfo in the following definition mean depth-first in, and depth-first out. Definition 8 defines the depth-first search of a graph first in terms of vertices as they enter the stack, and then also, respectively, as the vertices quit the stack.

Definition 8: A depth-first search in a graph G=(X,U) yields two total orderings of the vertices, namely: dfi(G) (resp. dfo(G)) which is , for every vertex x an element of X, defined by dfi(x) = i (resp. dfo(x) = i) if x enters (resp. quits) the stack of the depth-first search at the ith rank.

The following theorem relates depth-first search and dfgreedy linear extensions.

Theorem: Let P be a poset with a unique minimal element a, and G(a) be a diagram of P (i.e. the graph of the relation P with no transitivity arcs) then there is an isomorphism between the set of dfgreedy linear extensions of P and the set of dfo^d orders yielded by the depth-first searches on G(a) starting from the vertex a.

This yields the inheritance algorithm:

1. Perform a depth-first search of G(a), according to μ^d, starting from the vertex a.

2. $\wp_a = dfo^d(G(a))$

Remember from the beginning of our discussion of the inheritance search that superscript "d" represents the dual of a relation. Also, we have defined μ to be the maping of multiplicity in our inheritance search. I will use the graph from figure 8 as an example to show how the preceeding algorithm will result in the partial order that is the inheritance search. The partial order will be the dfgreedy linear extension given in figure 8.

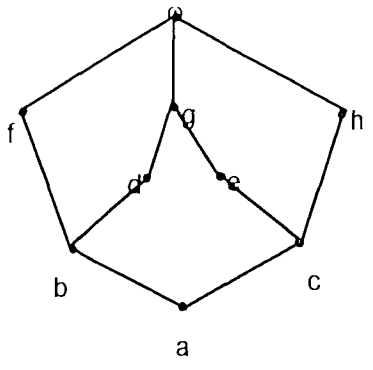
Please note that there is no multiplicity specified as part of the graph in figure 8. I don't feel that handling multiplicity is essential for an implementation of multiple inheritance. Ducournau and Habib acknowledge that it makes the algorithm more complex. In fact, they go into more depth about some problems that it can introduce, and how to solve them. The discussion of it here has been just to say what it is. I feel that a discussion of

how to do an inheritance search would be incomplete without acknowledging that such a facility could be implemented. However, it would probably be more practical to implement this feature by enforcing that the programer use compound selectors to specify which super-class he wants to inherit a property from, rather than specifying which super-class he prefers to inherit from first using multiplicity.

An example based on the graph in figure 8 follows. A depth-first greedy search produces the stack in figure 9. Each entry on the stack contains the name of the vertex that has been traversed, followed by the number of its children that have been popped from the stack. (The number of children that have been popped from the stack has no significance other than to illustrate more explicitly the order in which the vertices have been traversed on the graph.) The dfo(G(a)) is [ω,g,e,h,c,d,f,b,a]. This is the list of vertices as they are removed from the stack. The dual of this list, is the same list in reverse order. dfo_d = [a,b,f,d,c,h,e,g,ω] is the partial order that will be the path of the inheritance search for a property starting from object a.

The Stack

(a,0)
(a,0)(b,0)
(a,0)(b,0)(f,0)
(a,0)(b,0)(f,0)(d,0)
(a,0)(b,0)(f,0)(d,0)(c,0)
(a,0)(b,0)(f,0)(d,0)(c,0)(h,0)
(a,0)(b,0)(f,0)(d,0)(c,0)(h,0)(e,0)
(a,0)(b,0)(f,0)(d,0)(c,0)(h,0)(e,0)(g,0)
(a,0)(b,0)(f,0)(d,0)(c,0)(h,0)(e,0)(g,0)(ω,0)
(a,0)(b,0)(f,0)(d,0)(c,0)(h,0)(e,0)(g,1)
(a,0)(b,0)(f,0)(d,0)(c,0)(h,0)(e,1)
(a,0)(b,0)(f,0)(d,0)(c,1)(h,0)
(a,0)(b,0)(f,0)(d,0)(c,2)
(a,1)(b,0)(f,0)(d,0)
(a,1)(b,1)(f,0)
(a,1)(b,2)
(a,2)



$dfo = [\omega,g,e,h,c,d,f,b,a]$

$dfo_d = [a,b,f,d,c,h,e,g,\omega]$

Figure 9: Example inheritance search of graph in figure 8, using our proposed algorithm.

Name Collision

Singh [17] talks about "name collision" and "method combination" as problems that need to be solved. I need to do more research on method combination, but I feel it is really the same problem as name collision (or at least very closely related). Snyder [18] talks about method combination. He talks about universal methods (i.e., initialize) that are not inherited. This seems like much ado about nothing. I believe that method combination is not a serious problem. That, by putting the onus on the programmer to write more code if he wishes to combine methods from multiple ancestors, one can just not provide method combination as a feature. I don't believe that this will weaken the language. It will make its implementation and use more straight forward. Then it can just be treated as any other name collision when like named attributes are inherited. I'll ellaborate on Knudsen's [11] design to handle name collision.

What is Name Collision?

Knudsen answers the question of how to inherit attributes with identical names from multiple paths in a classification hierarchy. He describes the problem as how to decide how these multiple classification paths are reflected in the class being defined.

Some languages treat name collision as illegal. Others treat name collisions as separate declarations of equal right, while others treat name collisions as specialization of the attribute. Knudsen explains that there is no one simple inheritance mechanism to resolve name collision. He examines the underlying issues of how name collision can occur, and what the application designer may have in mind in defining classes with names of attributes

that are the same. He aims is to solve as many name collisions as possible at compile time, and to ensure the highest degree of polymorphism.

Horizontal vs. Vertical

The broadest classification of name collision comes from two different ways with which it can arise. Horizontal name collision occurs when a class inherits several attributes with the same name from different super-classes. Vertical name collision is when a class defines an attribute with the same name as one (or more) attributes inherited from one of its super-classes.

Issues of Name Collision

Knudsen describes three different views on the consequences of a name collision -- intended, casual, and illegal. These views are interpretations of what the application designer's intention was in using name collision and whether it can be resolved.

Name collision is intended if different attributes with the same name describe the same phenomenon. It is casual if different attributes with the same name describe different phenomena. And it is illegal if the relationship between attributes, names and phenomena must be unique.

Intended

Intended name collision implies that there is really one attribute, that will have several

specifications (one for each inherited attribute, and possibly one in the class itself). The several specifications together must constitute the full specification of the unique attribute. This will ensure the polymorphic property.

If, however, it is impossible to determine whether the several specifications together constitute the full specification of the attribute, it is not possible to guarantee polymorphism. If we have an intended vertical name collision, and we know that the specification of B.x is a specialization of the specification A.x, we can ensure the polymorphic property. If we have an intended horizontal name collision, the situation is more complex. If classes of A.x and B.x are super-classes of C.x, and A and B have a common super-class, then the two attributes are to some extent related. It might then be plausible to consider them as different views of the same attribute. And therefore assume that the polymorphic property is valid.

Casual

When a name collision is considered casual, we are allowing several attributes with the same name, and not necessarily (probably not) related specifications. If they are the same inherited specification, it doesn't matter. Knudsen treats such attributes as if they are different. In this case it is important to be able to distinguish between the different attributes by some means other than their names. Attributes are usually qualified with the name of the class from which they are inherited.

Illegal

In some cases, a name collision must be considered illegal. The relation between names, attributes, and phenomena must be unique. These will be caught at compile time.

The Need for Programmer Control

Knudsen points out that all three views on name collision are useful, and that each corresponds to different aspects of programming and modeling. Choosing one particular interpretation will result in the inability to express certain structures. The question then is how to determine which of these views is correct to implement at compile time. Knudsen implies that this can be accomplished with the help of "programmer control".

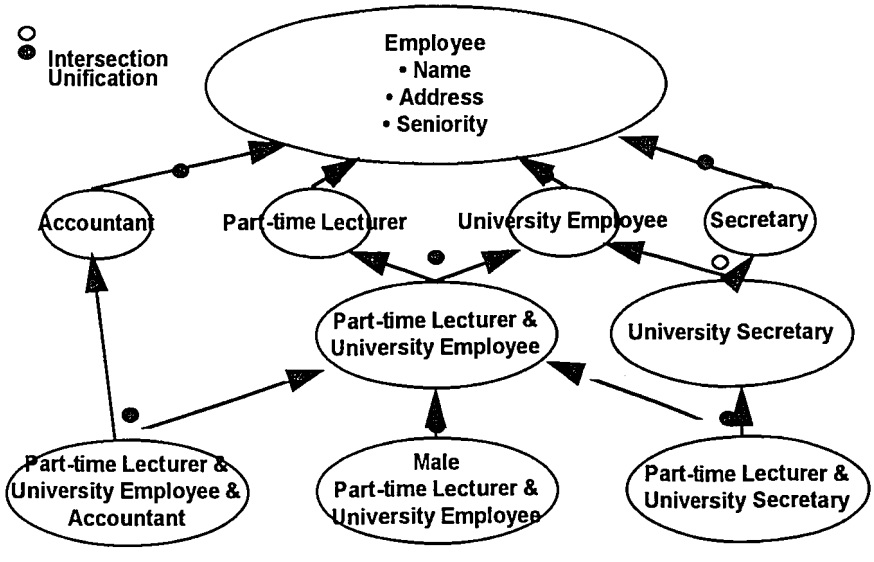


Figure 10: Multiple Inheritance with Name Collision.

He provides an example university employee multiple classification hierarchy that includes

three attributes -- Name, Address, and Seniority. Name and Address are what he calls "singleton" attributes (i.e., any instance of any class in the hierarchy will have only one Name and one Address). On the other hand, the Seniority attribute is concerned with the seniority of the person as employed in the particular job category. This might lead to specifying that the Seniority attribute should be inherited down the hierarchy with duplicates when multiple classification is involved. A part-time lecturer and university employee may have different seniority as a lecturer and as an university employee.

The Solution

Specialization Methods

Unification vs. Intersection

Contrary to the part-time lecturer and university employee who holds down two positions in the employee hierarchy, a university secretary inherits from both University Employee and Secretary, holding only one position. Knudsen uses these sub-classes as illustrations of two different specialization methods. The first specialization method he calls "unification". Unification takes care of the kind of specialization where the specialized class is supposed to model the unification of all the classes in its classification hierarchy. That is, if a horizontal name collision should occur, it should be treated as a casual horizontal name collision, giving rise to multiple attributes with the same name. The second specialization method (called intersection) takes care of the kind of specialization where the specialized class is supposed to model the intersection of all the classes in its classification hierarchy. That is, if a horizontal name collision occurs, and the attribute for all the immediate super-classes is inherited from one common super-class, then the name

collision is treated as an intended horizontal name collision. Now using the above rule, assuming the part-time lecturer and university employee is an example of unification inheritance, and assuming the university secretary is an example of intersection inheritance, will give us the correct interpretation of how seniority should be inherited.

Inheritance Properties of Attributes

Singleton vs. Plural

Besides specifying what specialization method (unification or intersection) is being used when creating a sub-class, Knudsen points out that we have to specify inheritance properties of individual attributes, too. In order to ensure that attributes such as Name and Address only exist in one copy in any future specializations of the class, they must be defined as "singleton" attributes. All other attributes are said to be plural (as is Seniority).

Discussion of Unification and Intersection Inheritance

Knudsen discusses in further detail unification and intersection inheritance and non-singleton attributes. I'd like to just go over two cases that appear to be salient.

Disjoint Multiple Classification

The first case (I.) is what he calls Disjoint Multiple Inheritance. This is when the two super-classes do not have a common parent. This is the place where we decide to consider some name collisions as being illegal.

I.(a) Unification

When disjoint hierarchies are combined using unification inheritance, we consider name collisions as being casual, and allow duplicate instances of attributes having the same name. The reason is that we want to combine two independent hierarchies. An example is combining a hierarchy concerning job type (teacher, secretary, trucker, etc.) with a hierarchy concerning nationality (Danish, Swedish, American, etc.). If there is an attribute X in both hierarchies, then this attribute will not be considered as being the same attribute (i.e., the two hierarchies using the same name by coincidence).

I.(b) Intersection

When hierarchies are combined using intersection inheritance, we are stressing that the involved hierarchies are considered as mutually contributing to the full specification of the new class. That is, the new class is created by merging attributes. In the case of name collision, we have to consider whether it makes sense to merge the attributes. If the attributes are not defined in a common super-class, then there is no way to ensure that the attributes are related in any way. Any automatic rule must consider such name collisions in disjoint intersection inheritance as "illegal" name collisions.

Simple Multiple Classification

The second case (II.) is what he calls Simple Multiple Classification. In this simple case of multiple inheritance, the classification hierarchies of the super-classes share a common super-class in which the attribute is defined (and no multiple inheritance is involved in the super-class hierarchies).

II.(a) Unification

This case is the same as disjoint unification above, giving rise to two X attributes in the sub-class.

II.(b) Intersection

In this case the super-class hierarchies share a common super-class in which the X attribute is defined, and it is therefore possible to assure that the inherited X attributes are related and thus it makes sense to merge them into one attribute.

Support for All Three Views

Knudsen concludes that his prime result is that all three views on name collision need to be supported in a programming language unless one accepts that it is not necessary to express certain structures. He does a nice job of laying out the arguments of what may have motivated the definition of attributes within classes in a multiple classification hierarchy resulting in a name collision, and discussing the alternatives for how name collisions can be viewed. He gives inspiration to potential designers of programming languages with support of classification hierarchies with multiple classification.

Conclusion

I started out by assuming that Multiple Inheritance is an important programming concept. Not everyone would agree; see Stroustrup [19]. Nevertheless, I proposed an approach to implementing multiple inheritance in an object-oriented programming language. Some basic object-oriented programming concepts were explained. Then some observations about how super-classes can be combined into sub-classes were made. This was done with an effort to preserve encapsulation in its purest form. I hope this gave a good background for how multiple inheritance should be used. Two important issues, that need to be resolved before one can implement multiple inheritance, were identified. They are the inheritance search, and name collision. These two issues were discussed in detail to provide the meat of the design required for an implementation. Armed with the ideas presented in this paper, one should have the foundation on which multiple inheritance can be built into an object-oriented programing language.

Bibliography

[1] D. G. Bobrow et al, "CommonLoops: Merging Lisp and Object Oriented Paradigm", Proc. ACM Conf. on Object Oriented Systems, Languages, and Applications, Sept 1986, Published as SIGPLAN Notices Nov. 1986 pp 17-29.

[2] A. Borning & D. Ingalls, "Multiple Inheritance in Smalltalk-80", Proc. of the National Conf. on A.I., 1982, pp 234-237.

[3] L. Cardelli, "A Semantics of Multiple Inheritance", Proc. of the Conf. on the Semantics of Data Types, Springer Vertag Lecture Notes in CS, June 1984.

[4] B. Carre & J. Geib, "The Point of View Notion for Multiple Inheritance", ECOOP/OOPSLA 90 Proceedings, Oct. 1990

[5] S. Cook, "OOPSLA '87 Panel P2: Varieties of Inheritance", OOPSLA 87 Addendum to the Proceedings, Oct. 1987

[6] B. Cox, "Object-Oriented Programming, An Evolutionary Approach", Addison-Wesley, 1986

[7] R. Ducournau & M. Habib, "On Some Algorithms for Multiple Inheritance in Object Oriented Programming", ECOOP 87, Paris

[8] R. Ducournau & M. Habib, et. al, "Proposal for a Monotonic Multiple Inheritance Linearization", OOPSLA 94 Proceedings, Oct. 1994

[9] A. Goldberg & D. Robson, "Smalltalk-80 The Language and its Implementation", Addison-Wesley, 1983

[10] N. Guimaraes, "Building Generic User Interface Tools: An Experience with Multiple Inheritance", OOPSLA 91, pp 89-96

[11] J. L. Knudsen, "Name Collision in Multiple Classification Hierarchies", ECOOP 88, Oslo, Aug. 1988, pp 94-109

[12] J. Martin, "Principles of Object-Oriented Analysis and Design", Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993

[13] B. Meyer, "Harnessing Multiple Inheritance", Journal of Object Oriented Programming, V. 1, No. 4, Nov./Dec. 1988, pp 48-51

[14] B. Meyer, "Object-Oriented Software Construction", Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988

[15] D. A. Moon, "Object Oriented Programming with Flavors", Proc. ACM Conf. on Object Oriented Systems, Languages, and Applications, Sept 1986, Published as SIGPLAN Notices Nov. 1986 pp 1-8

[16] C. Schaffert et al, "An Introduction to Trellis/Owl", Proc. ACM Conf. on Object Oriented Systems, Languages, and Applications, Sept 1986, Published as SIGPLAN Notices Nov. 1986 pp 9-16

[17] G. B. Singh, "Single Versus Multiple Inheritance in Object Oriented Programming" OOPS Messenger, V. 5, Issue 1, Jan. 1994, pp 34-43

[18] A. Snyder, "Inheritance and the Development of Encapsulated Software Systems", Research Directions in Object Oriented Programming, Edited by Bruce Shriver and Peter Wegner, MIT Press, 1987, pp 165-188

[19] B. Stroustrup, "Multiple Inheritance in C++", EUUG (European UNIX systems User Group) Spring '87 Proceedings on board M/S Mariella sailing between Helsinki and Stockholm, May 1987

[20] D. Touretzky, "Research Notes in AI. The Mathematics of Inheritance Systems", Pittman, London; Available in the Western Hemisphere from Morgan Kaufmann, Los Altos, 1986

END

OF

TITLE